Equivalence checking of sequential programs: models, algorithms, complexity

Vladimir A. Zakharov
Lomonosov Moscow State University
zakh@cs.msu.su
PART 1
Equivalence checking problem in general

PART 2
Equivalence checking problem in propositional model of sequential programs
There are 4 principal lines of research in formal analysis and verification of computer programs:

1. **Equivalence Checking**
   - A. A. Lyapunov, Y. I. Ianov: 1956

2. **Proof-Theoretic Approach**

3. **Static Analysis**

4. **Model Checking**
There are 4 principal lines of research in formal analysis and verification of computer programs:

1. **Equivalence Checking**
   A.A. Lyapunov, Y.I. Ianov: 1956

2. **Proof-Theoretic Approach**

3. **Static Analysis**

4. **Model Checking**
There are 4 principal lines of research in formal analysis and verification of computer programs:

3. **Static Analysis**

4. **Model Checking**
There are 4 principal lines of research in formal analysis and verification of computer programs:

2. **Proof-Theoretic Approach**  

3. **Static Analysis**  

4. **Model Checking**  
There are 4 principal lines of research in formal analysis and verification of computer programs:

1. **Equivalence Checking**
   A.A. Lyapunov, Y.I. Ianov: 1956

2. **Proof-Theoretic Approach**

3. **Static Analysis**

4. **Model Checking**
Two programs are equivalent if they have the same behaviour.

The equivalence checking problem is that of checking whether two given programs have the same behaviour,

or, in other words, a verification of a program against a specification which is another program.
By assigning various meanings to the terms «program», «same», «behaviour» we obtain a variety of equivalence checking problems:
By assigning various meanings to the terms «program», «same», «behaviour» we obtain a variety of equivalence checking problems:

**programs**: sequential, parallel, interactive, recursive, logic, distributed etc.
By assigning various meanings to the terms «program», «same», «behaviour» we obtain a variety of equivalence checking problems:

**Programs**: sequential, parallel, interactive, recursive, logic, distributed etc.

**Abstraction level**: propositional, first-order, high-order, etc.
By assigning various meanings to the terms «program», «same», «behaviour» we obtain a variety of equivalence checking problems:

**programs**: sequential, parallel, interactive, recursive, logic, distributed etc.

**abstraction level**: propositional, first-order, high-order, etc.

**behaviour**: trace, set of traces, function, relation, etc.
By assigning various meanings to the terms «program», «same», «behaviour» we obtain a variety of equivalence checking problems:

**Programs:** sequential, parallel, interactive, recursive, logic, distributed etc.

**Abstraction level:** propositional, first-order, high-order, etc.

**Behaviour:** trace, set of traces, function, relation, etc.
What is the use of equivalence checking?

1. Verification: digital integrated circuit designing, software upgrading.
2. Optimization: optimizing compilation.
5. Program understanding.
What is the use of equivalence checking?

1. **Verification**: digital integrated circuit designing, software upgrading.
2. **Optimization**: optimizing compilation.
3. **Refactoring**: clone detection.
4. **Security**: program obfuscation, malware code detection, plagiarism detection.
5. **Program understanding**.
VLSI designing route

Algorithmic Level (C++)

Transaction Level (System C)

Register Transfer Level (Verilog)

Logical Circuit Layout — Netlist (IEEE PDEF format)

Physical Layout (GDSII format)
VLSI designing route

Algorithmic Level (C++)

Transaction Level (System C)

Register Transfer Level (Verilog)

Logical Circuit Layout — Netlist (IEEE PDEF format)

Physical Layout (GDSII format)
Software upgrading

A considerable amount of software built in 70-80-s XX A.D. is still into the service. And it seems that nowadays nobody knows how these programs operate. But in the course of time it becomes more and more difficult to maintain such programs without a thorough enhancement.

What is more suitable:

- to build a completely new program which has the same functionality as the old one,
- or to equivalently transform the old program into a more refined one?
Program optimization and refactoring

At the beginning, mathematical models of programs and equivalence checking techniques were developed to prove the correctness of optimization transformations of computer programs.
Malware code detection

Polymorphic and metamorphic viruses modify their signature to avoid being detected every time when they infect new executables or replicate themselves.

The only characteristic which remains invariable for all generations of the same virus is their functionality (semantics).

Hence, the only way to detect for sure a metamorphic malicious code is to look for a pattern which has the same semantics as (i.e. equivalent to) some representative sample of the virus.
Program obfuscation

Program obfuscation is a semantic-preserving transformation aimed at bringing a program into such a form, which impedes the understanding of its algorithm and data structures or prevents extracting of some valuable information from the text of a program.

A secure obfuscation technique is a key component of every large-scale system of mobile computing (in particular, cloud computing).

An obfuscating transformation $\mathcal{O}$ is unsecure if given any pair of non-equivalent programs $\pi_1, \pi_2$ of the same size one can easily distinguish $\mathcal{O}(\pi_1)$ from $\mathcal{P}(\pi_2)$. 
This is a matter of a pure scientific curiosity: to what extent such artificial phenomenon as computer program is conceivable.

To understand (at the lowest level) the meaning of an object is to recognize whether two different objects have the same meaning or not.

Thus, equivalence checking problem for computer programs is the problem of program understanding.

Problem definition — O. Rabin, D. Scott \( (1959) \)

Solution — T. Harju, J. Karhumaki \( (1991) \)

Problem definition — O. Rabin, D. Scott (1959)

2. Equivalence checking problem for deterministic push-down automata.

Problem definition — S. Ginsburg, S. Greibach (1966)
Solution — G. Senizergues (1997)
Why semantic analysis of programs is so hard?

**Theorem [Rice–Uspensky]**

In every «natural» programming system every non-trivial functional property of programs is undecidable.
Why semantic analysis of programs is so hard?

**Theorem [Rice–Uspensky]**

In every «natural» programming system every non-trivial functional property of programs is undecidable.

«Natural» programming system has an effective interpreter and admits translation into it any other effective programming system.
Why semantic analysis of programs is so hard?

**Theorem [Rice–Uspensky]**

In every «natural» programming system every **non-trivial** functional property of programs is undecidable.

«Natural» programming system has an effective interpreter and admits translation into it any other effective programming system.

**Nontrivial** = non-empty and non-total.
Why semantic analysis of programs is so hard?

Theorem [Rice–Uspensky]

In every «natural» programming system every non-trivial functional property of programs is undecidable.

«Natural» programming system has an effective interpreter and admits translation into it any other effective programming system.

Nontrivial = non-empty and non-total.

Functional property refers only to the function (or relation) computed by a program.
Theorem [Rice–Uspensky]

In every «natural» programming system every non-trivial functional property of programs is undecidable.

Proof (Sketch) Reductio ad absurdum
Theorem [Rice–Uspensky]

In every «natural» programming system every non-trivial functional property of programs is undecidable.

Proof (Sketch) Reductio ad absurdum

Suppose that some non-trivial functional property $P$ has an effective decision procedure $Proc_P$. 
Theorem [Rice–Uspensky]

In every «natural» programming system every non-trivial functional property of programs is undecidable.

Proof (Sketch) Reductio ad absurdum

Suppose that some non-trivial functional property \( P \) has an effective decision procedure \( Proc_P \).

Consider an arbitrary interpreter of the programming system (a universal Turing machine) \( U \).

A pair of programs \( Proc_P \) and \( U \) can be combined into a new program \( \pi = \Psi(U, Proc_P) \) such that

\[
P(\pi) = true \iff P(\pi) = false.
\]
Why semantic analysis of programs is so hard?

Because it is difficult to analyse a behaviour of a Universal Turing Machine (UTM).

But how much complicated are UTMs?
Why semantic analysis of programs is so hard?

Because it is difficult to analyse a behaviour of a Universal Turing Machine (UTM).

But how much complicated are UTMs? They are very simple.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, B, L</td>
<td>2, A, R</td>
</tr>
<tr>
<td>1</td>
<td>2, A, R</td>
<td>2, B, L</td>
</tr>
<tr>
<td>2</td>
<td>1, A, R</td>
<td>0, A, L</td>
</tr>
</tbody>
</table>

This is the smallest weakly universal Turing machine
Why semantic analysis of programs is so hard?

Maybe, the complexity of a program is not in its size but in the topology of its control flow. Then the simplest programs have no embedded loops, i.e. the nesting depth of all cycles in a program is at most 1.
Why semantic analysis of programs is so hard?

Maybe, the complexity of a program is not in its size but in the topology of its control flow. Then the simplest programs have no embedded loops, i.e. the nesting depth of all cycles in a program is at most 1.

**Theorem [Yanov]**

Programs without embedded loops over the basis of instructions

\[
x=x+1; \quad x=0; \quad x==y
\]

compute all functions expressible in Presburger Arithmetic
Why semantic analysis of programs is so hard?

Presburger Arithmetic (PA) is a formal theory of non-negative integers with addition but without multiplication. Such functions and predicates as

\[ x + y, \ x - y, \ x = y \mod n \]

are expressible in PA, whereas

\[ x \times y, \ x = y \mod z \]

are not.
Why semantic analysis of programs is so hard?

Presburger Arithmetic (PA) is a formal theory of non-negative integers with addition but without multiplication. Such functions and predicates as

\[ x + y \, , \, x - y \, , \, x = y \mod n \]

are expressible in PA, whereas

\[ x \times y \, , \, x = y \mod z \]

are not.

PA is a decidable theory but the lower bound of its worst case complexity is \(2^{2^{cn}}\).
Why semantic analysis of programs is so hard?

If there are still any doubts then consider a pair of programs

function $\pi_1$ (x: positive integer)
        A[0]=0; A[1]=0; A[2]=x; i=2;
        do
            else A[i+1]=3*A[i]+1;
            i=i+1
        until A[i]==A[i-3]
        return A[i]
end

function $\pi_2$ (x: positive integer)
        if (x==2) \lor (x==4) then return x
        else return 1
end
Why semantic analysis of programs is so hard?

It is a generally accepted opinion, that $\pi_1$ and $\pi_2$ are equivalent (compute the same function) but nobody yet can prove this assertion for sure.

The sequence $A[0], A[1], A[2], \ldots$ from the program $\pi_1$ is the famous Collatz sequence which always (hypothetically) tends to the cycle 4, 2, 1.
Why semantic analysis of programs is so hard?

It is a generally accepted opinion, that $\pi_1$ and $\pi_2$ are equivalent (compute the same function) but nobody yet can prove this assertion for sure.

The sequence $A[0], A[1], A[2], \ldots$ from the program $\pi_1$ is the famous Collatz sequence which always (hypothetically) tends to the cycle $4, 2, 1$.

That is why

«Even simple programs are hard to analyze»

Resume

One can not expect to build an efficient equivalence checking procedure which yields a result for every pair of programs (even for «simple» programs).
Resume

One can not expect to build an efficient equivalence checking procedure which yields a result for every pair of programs (even for «simple» programs).

Nevertheless, we may hope to find a technique which could check the equivalence for some pairs of «similar» programs.

But what to begin with?
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a) \text{ and } a \otimes a.\]
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a)\] and \[a \otimes a\].

Then

\[(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c)\]

if \(\otimes\) is commutative: \(c \otimes a = a \otimes c\)
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a) \text{ and } a \otimes a.\]

Then

\[(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c)\]

if \( \otimes \) is distributive w.r.t. \( \oplus \): 

\[(a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c)\]
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a)\] \text{and} \[(a \otimes a)\].

Then

\[(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c) = a \otimes 0\]

if \(b\) and \(c\) is a reciprocal pair w.r.t. \(\oplus\): \(b \oplus c = 0\).
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a)\] and \[a \otimes a\].

Then

\[(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c) = a \otimes 0 = a\]

if 0 is a unit element w.r.t. \(\otimes\): \[a \otimes 0 = a\]
Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a) \text{ and } a \otimes a.\]

Then

\[
(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c) = \\
= a \otimes 0 = a = a \otimes a
\]

if \(\otimes\) is idempotent: \(a = a \otimes a\)
Algebraic example

Suppose we need to prove the equivalence of two algebraic expressions

\[(a \otimes b) \oplus (c \otimes a) \text{ and } a \otimes a.\]

Then

\[(a \otimes b) \oplus (c \otimes a) = (a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c) = a \otimes 0 = a = a \otimes a\]

So, we prove the equivalence of these expressions by relying entirely upon some algebraic laws and ignoring such tiresome things as "domain", "interpretation", etc.

Can we play the same trick with programs?
Algebraic example

Et pourquoi pas?

Instructions $a : x = f(y)$ and $b : z = g(u, v)$ commute:
$(a; b) \equiv (b; a)$

Instructions $a : x = x + 1$ and $b : x = x - 1$ are reciprocal:
$(a; b) \equiv (b; a) \equiv 1$

Instruction $b : x = f(y)$ absorbs $a : x = g(x, y)$:
$(a; b) \equiv b$

Instruction $0_c : x = c$ is a right zero for any instruction of the form $a : x = t(\cdots)$:
$(a; 0_c) \equiv 0_c$
Algebraic example

To develop this idea we could

1. introduce the concept of sequential imperative programs at the propositional abstraction level by considering basic instructions and tests (predicates) as unstructured entities (letters);
Algebraic example

To develop this idea we could

1. introduce the concept of sequential imperative programs at the propositional abstraction level by considering basic instructions and tests (predicates) as unstructured entities (letters);

2. define the semantics of these programs in such a way that various interpretations of basic instructions may be specified by the sets of algebraic equalities of the form $a_1; a_2; \ldots a_n = b_1; b_2; \ldots b_m$;
Algebraic example

To develop this idea we could

1. introduce the concept of sequential imperative programs at the propositional abstraction level by considering basic instructions and tests (predicates) as unstructured entities (letters);

2. define the semantics of these programs in such a way that various interpretations of basic instructions may be specified by the sets of algebraic equalities of the form
   \( a_1; a_2; \ldots a_n = b_1; b_2; \ldots b_m; \)

3. and, finally, study how the algebraic properties of basic instructions affect the complexity of equivalence checking problem.
END OF PART 1
This talk is about:

1. Propositional models of sequential programs: syntax and semantics.
2. Equivalence checking problem.
3. Ordered frames and multi-tape machines.
4. Equivalence checking in polynomial time.
5. Other complexity issues.
Programs: syntax

Alphabet

\[ A = \{a_1, a_2, \ldots, a_m\} \] — a set of basic instructions (statements);
Programs: syntax

Alphabet

\[ A = \{a_1, a_2, \ldots, a_m\} \text{ — a set of basic instructions (statements)} ; \]
\[ \mathcal{P} = \{p_1, p_2, \ldots, p_k\} \text{ — a set of basic predicates} ; \]
Alphabet

\[ A = \{ a_1, a_2, \ldots, a_m \} \] — a set of basic instructions (statements) ;

\[ P = \{ p_1, p_2, \ldots, p_k \} \] — a set of basic predicates ;

a condition is a binary tuple \[ \Delta = \langle \delta_1, \delta_2, \ldots, \delta_k \rangle \] of truth values of basic predicates;

\[ C = \{ \Delta_1, \Delta_2, \ldots, \Delta_{2k} \} \] — the set of conditions.
Programs: syntax

Program

Program \( \pi = \langle V, \text{entry}, \text{exit}, T, B \rangle \) is a finite labeled transition system, where

- \( V \) — a non-empty set of program points;
Programs: syntax

Program

Program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ is a finite labeled transition system, where

- $V$ — a non-empty set of program points;
- $\text{entry}$ — the starting point;
Programs: syntax

Program

Program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ is a finite labeled transition system, where

- $V$ — a non-empty set of program points;
- entry — the starting point;
- exit — the final point;
Programs: syntax

Program

Program \( \pi = \langle V, \text{entry}, \text{exit}, T, B \rangle \) is a finite labeled transition system, where

- \( V \) — a non-empty set of program points;
- \( \text{entry} \) — the starting point;
- \( \text{exit} \) — the final point;
- \( T : (V - \{\text{exit}\}) \times C \rightarrow V \) — a transition function;
Programs: syntax

Program

Program \( \pi = \langle V, \text{entry}, \text{exit}, T, B \rangle \) is a finite labeled transition system, where

- \( V \) — a non-empty set of program points;
- \( \text{entry} \) — the starting point;
- \( \text{exit} \) — the final point;
- \( T : (V \setminus \{\text{exit}\}) \times C \to V \) — a transition function;
- \( B : (V \setminus \{\text{exit}\}) \to A \) — a binding function.
Consider, for example, a program

```
function \( \pi(x, y) \)
    while \( x < 0 \) do \( x = x + y \) od;
    do \( y = y + x \) until \( y > x \);
end
```

Basic instructions: \( a_1 : x = x + y, \quad a_2 : y = y + x \).

Basic predicates: \( p_1 : x < 0, \quad p_2 : y > x \).
Programs: syntax

The same program $\pi$ as a flow-chart
Programs: syntax

The same $\pi$ as a propositional sequential program

\[
\begin{align*}
\langle 1, 0 \rangle & \rightarrow a_1 \langle 1, 1 \rangle \\
\langle 1, 1 \rangle & \rightarrow v_1 \langle 1, 0 \rangle \\
\langle 0, 0 \rangle & \rightarrow a_2 \langle 0, 1 \rangle \\
\langle 0, 1 \rangle & \rightarrow v_2 \langle 0, 0 \rangle \\
\langle 0, 0 \rangle & \rightarrow \varepsilon \langle 0, 1 \rangle \\
\langle 0, 1 \rangle & \rightarrow \langle 1, 1 \rangle \\
\langle 1, 1 \rangle & \rightarrow \text{exit}
\end{align*}
\]
Programs: semantics

Data space

Dynamic logic frame $\mathcal{F} = \langle S, s_0, R \rangle$ provides an interpretation to basic instructions:

- $S$ — an non-empty set of data states,
Programs: semantics

Data space

Dynamic logic frame \( \mathcal{F} = \langle S, s_0, R \rangle \) provides an interpretation to basic instructions:

- \( S \) — an non-empty set of data states ,
- \( s_0 \) — the initial data state , \( s_0 \in S \),
Programs: semantics

Data space

Dynamic logic frame \( \mathcal{F} = \langle S, s_0, R \rangle \) provides an interpretation to basic instructions:

- \( S \) — an non-empty set of data states,
- \( s_0 \) — the initial data state, \( s_0 \in S \),
- \( R : \mathcal{A} \times S \rightarrow S \) — an updating function.
Programs: semantics

Data space

Dynamic logic frame $\mathcal{F} = \langle S, s_0, R \rangle$ provides an interpretation to basic instructions:

- $S$ — an non-empty set of data states,
- $s_0$ — the initial data state, $s_0 \in S$,
- $R : A \times S \rightarrow S$ — an updating function.

For every finite sequence of basic instructions $h = b_1, \ldots, b_{n-1}, b_n$ we write $R^*(h, s)$ for the composition $R(b_n, R(b_{n-1}, \ldots, R(b_1, s) \ldots))$. 
Programs: semantics

Data space

Dynamic logic frame $\mathcal{F} = \langle S, s_0, R \rangle$ provides an interpretation to basic instructions:

- $S$ — an non-empty set of data states,
- $s_0$ — the initial data state, $s_0 \in S$,
- $R : \mathcal{A} \times S \rightarrow S$ — an updating function.

For every finite sequence of basic instructions $h = b_1, \ldots, b_{n-1}, b_n$ we write $R^*(h, s)$ for the composition $R(b_n, R(b_{n-1}, \ldots, R(b_1, s) \ldots))$.

Dynamic logic structure $M = (\mathcal{F}, \xi)$ provides an interpretation to basic predicates:

- $\mathcal{F}$ — a frame,
- $\xi : S \rightarrow C$ — an evaluation function.
Programs: syntax and semantics

Dynamic logic frame

\[ A = \{a, b\} \]

\( a: x = f(x) \)

\( b: x = g(x) \)

Free semigroup (monoid)
Programs: syntax and semantics

Dynamic logic frame

$A = \{a, b\}$

$a$: $x = f(x)$

$b$: $y = g(y)$

Free commutative semigroup (monoid)
Programs: syntax and semantics

Dynamic logic frame

\[ A = \{ a, b, c \} \]
\[ a: x = f(x) \]
\[ b: y = g(y) \]
\[ c: \{ x = c_1; y = c_2 \} \]

Semantics of programs with commutative statements and a reset instruction

Free commutative semigroup with a right zero
Program run

A run of a program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ on a dynamic structure $M = \langle S, s_0, R, \xi \rangle$ is a sequence of pairs

$$
\text{comp}(\pi, M) = (v_0, s_0), (v_1, s_1), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots
$$

such that

1. $s_0$ is the initial state of $M$, $v_0 = \text{entry}$;
2. $v_{i+1} = T(v_i, \xi(s_i))$ and $s_{i+1} = R(s_i, B(v_i))$ hold for every $i \geq 1$;
3. if $\text{comp}(\pi, M)$ ends with a pair $(v_N, s_N)$ then $v_N = \text{exit}$.

If a run $\text{comp}(\pi, M)$ ends with a pair $(\text{exit}, s_N)$ then the result of the run is $\text{comp}(\pi, M) = s_N$.

If a run $\text{comp}(\pi, M)$ is infinite then the result of the run $\text{comp}(\pi, M)$ is undefined.
Program run

A run of a program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ on a dynamic structure $M = \langle S, s_0, R, \xi \rangle$ is a sequence of pairs

$$comp(\pi, M) = (v_0, s_0), (v_1, s_1), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots$$

such that

1. $s_0$ is the initial state of $\mathcal{F}$, $v_0 = \text{entry}$;
Programs: syntax and semantics

Program run

A run of a program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ on a dynamic structure $M = \langle S, s_0, R, \xi \rangle$ is a sequence of pairs

$$\text{comp}(\pi, M) = (v_0, s_0), (v_1, s_1), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots$$

such that

1. $s_0$ is the initial state of $\mathcal{F}$, $v_0 = \text{entry}$;
2. $v_{i+1} = T(v_i, \xi(s_i))$ and $s_{i+1} = R(s_i, B(v_i))$ hold for every $i \geq 1$;
Program run

A run of a program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ on a dynamic structure $M = \langle S, s_0, R, \xi \rangle$ is a sequence of pairs

$$\text{comp}(\pi, M) = (v_0, s_0), (v_1, s_1), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots$$

such that

1. $s_0$ is the initial state of $F$, $v_0 = \text{entry}$;
2. $v_{i+1} = T(v_i, \xi(s_i))$ and $s_{i+1} = R(s_i, B(v_i))$ hold for every $i \geq 1$;
3. if $\text{comp}(\pi, M)$ ends with a pair $(v_N, s_N)$ then $v_N = \text{exit}$.
Program run

A run of a program $\pi = \langle V, \text{entry}, \text{exit}, T, B \rangle$ on a dynamic structure $M = \langle S, s_0, R, \xi \rangle$ is a sequence of pairs

$$\text{comp}(\pi, M) = (v_0, s_0), (v_1, s_1), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots$$

such that

1. $s_0$ is the initial state of $\mathcal{F}$, $v_0 = \text{entry}$;
2. $v_{i+1} = T(v_i, \xi(s_i))$ and $s_{i+1} = R(s_i, B(v_i))$ hold for every $i \geq 1$;
3. if $\text{comp}(\pi, M)$ ends with a pair $(v_N, s_N)$ then $v_N = \text{exit}$.

If a run $\text{comp}(\pi, M)$ ends with a pair $(\text{exit}, s_N)$ then the result of the run is $[\text{comp}(\pi, M)] = s_N$.

If a run $\text{comp}(\pi, M)$ is infinite then the result of the run $[\text{comp}(\pi, M)]$ is undefined.
The equivalence checking problem

Programs $\pi_1$ and $\pi_1$ are called equivalent on a dynamic structure $M$ ($\pi_1 \sim_M \pi_2$ in symbols) iff $[comp(\pi_1, M)] = [comp(\pi_2, M)]$. 
The equivalence checking problem

Programs \( \pi_1 \) and \( \pi_1 \) are called equivalent on a dynamic structure \( M \) (\( \pi_1 \sim_M \pi_2 \) in symbols) iff \([\text{comp}(\pi_1, M)] = [\text{comp}(\pi_2, M)]\).

Programs \( \pi_1 \) and \( \pi_1 \) are called equivalent on a dynamic frame \( F \) (\( \pi_1 \sim_F \pi_2 \) in symbols) iff \( \pi_1 \sim_M \pi_2 \) holds for every dynamic structure \( M = (F, \xi) \).
The equivalence checking problem

Programs $\pi_1$ and $\pi_1$ are called equivalent on a dynamic structure $M$ ($\pi_1 \sim_M \pi_2$ in symbols) iff $[\text{comp}(\pi_1, M)] = [\text{comp}(\pi_2, M)]$.

Programs $\pi_1$ and $\pi_1$ are called equivalent on a dynamic frame $\mathcal{F}$ ($\pi_1 \sim_{\mathcal{F}} \pi_2$ in symbols) iff $\pi_1 \sim_M \pi_2$ holds for every dynamic structure $M = (\mathcal{F}, \xi)$.

Equivalence checking problem for a frame $\mathcal{F}$: given a pair of programs $\pi_1$ and $\pi_2$ check whether $\pi_1 \sim_{\mathcal{F}} \pi_2$. 
The equivalence checking problem

- The earliest paper on formal methods in computer science: the introducing of Yanov schemes (A. Lyapunov, Y. Yanov, 1957)
The equivalence checking problem

- The earliest paper on formal methods in computer science: the introducing of Yanov schemes (A. Lyapunov, Y. Yanov, 1957)
- Development of relationships between Yanov schemes, automata and computer programs (A. Ershov, V. Glushkov)
The equivalence checking problem

- The earliest paper on formal methods in computer science: the introducing of Yanov schemes (A. Lyapunov, Y. Yanov, 1957)
- Development of relationships between Yanov schemes, automata and computer programs (A. Ershov, V. Glushkov)
The equivalence checking problem

- The earliest paper on formal methods in computer science: the introducing of Yanov schemes (A. Lyapunov, Y. Yanov, 1957)
- Development of relationships between Yanov schemes, automata and computer programs (A. Ershov, V. Glushkov)

Open questions

1. How to build efficient (polynomial time) decision procedures?
The equivalence checking problem

- The earliest paper on formal methods in computer science: the introducing of Yanov schemes (A. Lyapunov, Y. Yanov, 1957)
- Development of relationships between Yanov schemes, automata and computer programs (A. Ershov, V. Glushkov)

Open questions

1. How to build efficient (polynomial time) decision procedures?
2. What are the algebraic properties of program statements that make the equivalence checking problem untractable/hard?
The equivalence checking problem

Universal (Turing-complete) programming systems

- Undecidable cases
- Decidable cases

Yanov schemes
The equivalence checking problem

Universal (Turing-complete) programming systems

Undecidable cases

Decidable cases

Untractable cases

Tractable cases

Yanov schemes
Equivalence checking in polynomial time

The idea of decision procedure

How to check the equivalence of two Deterministic Finite Automata (DFAs) $A_1, A_2$?
Equivalence checking in polynomial time

The idea of decision procedure

How to check the equivalence of two Deterministic Finite Automata (DFAs) $A_1$, $A_2$?

Build a product $A_0 = A_1 \otimes A_2$ and check the emptiness of $L(A_0)$. The DFA $A_0$ synchronizes the runs of $A_1$ and $A_2$; it accepts iff one of the DFAs $A_1$, $A_2$ accepts whereas the other rejects.
Equivalence checking in polynomial time

The idea of decision procedure

How to check the equivalence of two Deterministic Finite Automata (DFAs) $A_1$, $A_2$?

Build a product $A_0 = A_1 \otimes A_2$ and check the emptiness of $L(A_0)$. The DFA $A_0$ synchronizes the runs of $A_1$ and $A_2$; it accepts iff one of the DFAs $A_1$, $A_2$ accepts whereas the other rejects.

To check the emptiness of $L(A_0)$ one needs only to check the reachability of an accepting state of $A_0$ (which is NLOG-complete problem in the case of DFAs).
Equivalence checking in polynomial time

The idea of decision procedure

But how to synchronize the runs of programs $\pi_1, \pi_2$ (that can be viewed as DFAs) operating on a dynamic frame $\mathcal{F}$?
Equivalence checking in polynomial time

The idea of decision procedure

But how to synchronize the runs of programs $\pi_1, \pi_2$ (that can be viewed as DFAs) operating on a dynamic frame $\mathcal{F}$?

We need some synchronizer $D_{\mathcal{F}}$ which is

- able to detect whether runs of $\pi_1$ and $\pi_2$ reach the same intermediate data state $s$ in $\mathcal{F}$, and
Equivalence checking in polynomial time

The idea of decision procedure

But how to synchronize the runs of programs $\pi_1, \pi_2$ (that can be viewed as DFAs) operating on a dynamic frame $\mathcal{F}$?

We need some synchronizer $D_{\mathcal{F}}$ which is

- able to detect whether runs of $\pi_1$ and $\pi_2$ reach the same intermediate data state $s$ in $\mathcal{F}$, and
- very simple device to avoid untractability of the emptiness checking problem.
Equivalence checking in polynomial time

The idea of decision procedure

But how to synchronize the runs of programs $\pi_1, \pi_2$ (that can be viewed as DFAs) operating on a dynamic frame $\mathcal{F}$?

We need some synchronizer $D_{\mathcal{F}}$ which is

- able to detect whether runs of $\pi_1$ and $\pi_2$ reach the same intermediate data state $s$ in $\mathcal{F}$, and
- very simple device to avoid untractability of the emptiness checking problem.

Fortunately, such synchronizer $D_{\mathcal{F}}$ exists when $\mathcal{F}$ is an ordered frame.
Ordered frames and 2-tape machines

Ordered frames

Let \( F = \langle S, s_0, R \rangle \) be a frame. We say that a data state \( s'' \) is reachable from a data state \( s' \) (\( s' \preceq_F s'' \) in symbols) if \( R^*(h, s') = s'' \) holds for some sequence \( h \) of basic statements.
Ordered frames and 2-tape machines

Ordered frames

Let $\mathcal{F} = \langle S, s_0, R \rangle$ be a frame. We say that a data state $s''$ is reachable from a data state $s'$ ($s' \preceq_{\mathcal{F}} s''$ in symbols) if $R^*(h, s') = s''$ holds for some sequence $h$ of basic statements.

A frame $\mathcal{F}$ is called an ordered frame if $\preceq_{\mathcal{F}}$ is a partial order relation.

$\mathcal{F}$ is an ordered frame $\iff$ none of the data states of $\mathcal{F}$ can be achieved twice along any run of any program.
Ordered frames and 2-tape machines

Ordered frames

Let $\mathcal{F} = \langle S, s_0, R \rangle$ be a frame. We say that a data state $s''$ is reachable from a data state $s'$ ($s' \preceq_{\mathcal{F}} s''$ in symbols) if $R^*(h, s') = s''$ holds for some sequence $h$ of basic statements.

A frame $\mathcal{F}$ is called an ordered frame if $\preceq_{\mathcal{F}}$ is a partial order relation.

$\mathcal{F}$ is an ordered frame $\iff$ none of the data states of $\mathcal{F}$ can be achieved twice along any run of any program.

We show that the equivalence checking problem for ordered frames can be solved in polynomial time with the help of two-tape deterministic machines as synchronizers.
Ordered frames and 2-tape machines

Two-tape deterministic machines

Given a frame $\mathcal{F} = \langle S, s_0, R \rangle$ we introduce an equivalence relation $\approx_{\mathcal{F}}$ on the set $A^*$ of finite sequences of statements:

$$h_1 \approx_{\mathcal{F}} h_2 \iff R^*(h_1, s_0) = R^*(h_2, s_0)$$
Ordered frames and 2-tape machines

Two-tape deterministic machines

Given a frame $F = \langle S, s_0, R \rangle$ we introduce an equivalence relation $\approx_F$ on the set $A^*$ of finite sequences of statements:

$$h_1 \approx_F h_2 \iff R^*(h_1, s_0) = R^*(h_2, s_0)$$

A synchronizer is intended for detecting $h_1 \approx_F h_2$.

Since $\approx_F$ is a binary relation on $A^*$ this can be accomplished by a machine operating on two tapes in the same way as ordinary one-tape automata recognize languages (unary relations on $A^*$).
Ordered frames and 2-tape machines

Two-tape deterministic machines

A two-tape one-way deterministic machine (2-DM, for short) is a 6-tuple $D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle$, where

- $\Sigma$ is a finite input alphabet,
- $Q_1$ and $Q_2$ are two disjoint countable sets of internal states,
- $q_0$ is a distinguished initial state, $q_0 \in Q_1 \cup Q_2$,
- $F$ is a set of accepting states, $F \subseteq Q_1 \cup Q_2$,
- $\varphi : (Q_1 \cup Q_2) \times \Sigma \rightarrow Q_1 \cup Q_2$ is a (partial) transition function.
Ordered frames and 2-tape machines

Two-tape deterministic machines

A two-tape one-way deterministic machine (2-DM, for short) is a 6-tuple $D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle$, where

- $\Sigma$ is a finite input alphabet,
- $Q_1$ and $Q_2$ are two disjoint countable sets of internal states,
- $q_0$ is a distinguished initial state, $q_0 \in Q_1 \cup Q_2$,
- $F$ is a set of accepting states, $F \subseteq Q_1 \cup Q_2$,
- $\varphi : (Q_1 \cup Q_2) \times \Sigma \rightarrow Q_1 \cup Q_2$ is a (partial) transition function.
Ordered frames and 2-tape machines

Two-tape deterministic machines

A binary relation \( L_D \subseteq \Sigma^* \times \Sigma^* \) recognized by a 2-DM \( D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle \) is defined as follows:

\[
L_D = \{ \langle w_1, w_2 \rangle : w_1, w_2 \in \Sigma^*, \varphi^*(q_0, w_1, w_2) \in F \}.
\]
Ordered frames and 2-tape machines

Two-tape deterministic machines

A binary relation $L_D \subseteq \Sigma^* \times \Sigma^*$ recognized by a 2-DM $D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle$ is defined as follows:

$$L_D = \{ \langle w_1, w_2 \rangle : w_1, w_2 \in \Sigma^*, \varphi^*(q_0, w_1, w_2) \in F \}.$$

We say that a 2-DM $\mathcal{F}$ specifies a frame $F$ if $L_D = \approx F$. 
Ordered frames and 2-tape machines

Two-tape deterministic machines

A binary relation $L_D \subseteq \Sigma^* \times \Sigma^*$ recognized by a 2-DM $D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle$ is defined as follows:

$$L_D = \{ \langle w_1, w_2 \rangle : w_1, w_2 \in \Sigma^*, \varphi^*(q_0, w_1, w_2) \in F \}.$$

We say that a 2-DM $\mathcal{F}$ specifies a frame $F$ if $L_D = \mathord{\approx}_F$.

Theorem 1. A dynamic frame $\mathcal{F}$ is specified by a 2-DM iff $\mathcal{F}$ is an ordered frame.
Equivalence checking in polynomial time

Compound machines

Let $\mathcal{F}$ be an ordered frame specified by a 2-DM $D$, and $\pi', \pi''$ be a pair of programs to be checked. Following the basic idea we build the compound 2-DM $K_{\pi', \pi'', D_\mathcal{F}} = \pi' \otimes D_\mathcal{F} \otimes \pi''$. 

\[ (a'_1, \Delta'_1)(a'_2, \Delta'_2)(a'_3, \Delta'_3) \]

\[ (a''_1, \Delta''_1)(a''_2, \Delta''_2)(a''_3, \Delta''_3)(a''_4, \Delta''_4) \]
Equivalence checking in polynomial time

Compound machines

Let $\mathcal{F}$ be an ordered frame specified by a 2-DM $D$, and $\pi', \pi''$ be a pair of programs to be checked. Following the basic idea we build the compound 2-DM $K_{\pi',\pi'',D_{\mathcal{F}}} = \pi' \otimes D_{\mathcal{F}} \otimes \pi''$. 

\[
\begin{align*}
\pi' &\quad \pi'' \\
\downarrow &\quad \downarrow \\
(a'_1, \Delta'_1)(a'_2, \Delta'_2)(a'_3, \Delta'_3) &\quad (a''_1, \Delta''_1)(a''_2, \Delta''_2)(a''_3, \Delta''_3)(a''_4, \Delta''_4)
\end{align*}
\]
Equivalence checking in polynomial time

Compound machines

Let $\mathcal{F}$ be an ordered frame specified by a 2-DM $D$, and $\pi', \pi''$ be a pair of programs to be checked. Following the basic idea we build the compound 2-DM $K_{\pi', \pi'', D_{\mathcal{F}}} = \pi' \otimes D_{\mathcal{F}} \otimes \pi''$.
Equivalence checking in polynomial time

Compound machines

Theorem 2. Programs $\pi'$ and $\pi''$ are equivalent on a frame $F$ specified by a 2-DM $D$ iff the compound 2-DM $\hat{K} = K_{\pi',\pi'',D}$ complies with two following requirements:
Equivalence checking in polynomial time

Compound machines

**Theorem 2.** Programs $\pi'$ and $\pi''$ are equivalent on a frame $\mathcal{F}$ specified by a 2-DM $D$ iff the compound 2-DM $\hat{K} = K_{\pi',\pi'',D}$ complies with two following requirements:

A: $L_{\hat{K}} = \emptyset$, 
Equivalence checking in polynomial time

Compound machines

Theorem 2. Programs $\pi'$ and $\pi''$ are equivalent on a frame $\mathcal{F}$ specified by a 2-DM $D$ iff the compound 2-DM $\hat{K} = K_{\pi',\pi'',D}$ complies with two following requirements:

A: $L_{\hat{K}} = \emptyset$,

B: if $\hat{K}$ runs for an infinitely long time then $\hat{K}$ reads infinitely often on both tapes.
Equivalence checking in polynomial time

2-DMs with finitely many accepting states

**Theorem 3.** If a frame \( \mathcal{F} \) is specified by a 2-DM
\[
D = \langle \Sigma, Q_1, Q_2, q_0, F, \varphi \rangle
\]
such that the set of accepting states \( F \) is finite then the equivalence checking problem \( \pi_1 \sim_{\mathcal{F}} \pi_2 \) is in \( \text{NPTIME}^{O(D)} \).
Equivalence checking in polynomial time

**Injective frames**

A frame \( \mathcal{F} = \langle S, s_0, R \rangle \) is injective if every basic instruction computes an injective function, i.e., for every basic instruction \( a \) and for every pair of data states \( s_1, s_2 \)

\[
R(a, s_1) = R(a, s_2) \iff s_1 = s_2.
\]

**Theorem 4.** If an ordered frame \( \mathcal{F} \) is injective and specified by a 2-DM \( D \) then the equivalence checking problem \( \pi_1 \sim_{\mathcal{F}} \pi_2 \) is in \( \text{PTIME}^{O(D)} \).
Equivalence checking in polynomial time

Finite state 2-DMs

**Theorem 5.** If a frame $\mathcal{F}$ is specified by a 2-tape deterministic finite state automaton $D$ then the equivalence checking problem $\pi_1 \sim_{\mathcal{F}} \pi_2$ is in NLOGSPACE.
Equivalence checking in polynomial time

Finite state 2-DMs

Example 1.

Let $CE \subseteq A \times A \times A$.

The frame $F_{CE} = \langle S, s_0, R \rangle$ captures the effect of conditional equivalence of program statements if $R^*(ab, s) = R^*(ac, s)$ holds for every triple $(a, b, c) \in CE$ and a data state $s$.

This effect takes place when, say, $a, b, c$ are program statements $x:=1$, $y:=y+1$, and $y:=x+y$.

It is but a simple exercise to ascertain that for every set of triples $CE$ the frame $F_{CE}$ is specified by a 2-tape deterministic finite state automaton.
Finite state 2-DMs

Example 2.

We say that a instruction \( a \) absorbs an instruction \( b \) if \( b; a \equiv a \), i.e. instruction \( a \) undo the preceding instruction \( b \). This effect takes place when, say, \( a \) is \( x := z + y \) and \( b \) is \( x := x + y \).

Let \( H \) be an arbitrary set of pairs of basic instructions. The frame \( \mathcal{F}_H = \langle S, s_0, R \rangle \) captures the effect of \( H \)-absorption of basic instructions if \( R^*(s, ab) = R^*(s, b) \) holds for every data state \( s \in S \) and every pair \( a, b : (a, b) \in H^* \).

If a set of pairs of basic instructions \( H \) is such that the binary relation \( H^* \) is irreflexive then the frame \( \mathcal{F}_H \) is ordered and can be specified by a finite 2-DM.
Equivalence checking in polynomial time

Infinite state 2-DMs

Example 3.

The frame $\mathcal{F}_{\text{comm}} = \langle S, s_0, R \rangle$ captures the effect of commutativity of program statements if $R^*(s, ab) = R^*(s, ba)$ holds for every data state $s \in S$ and every pair $a, b$ of statements.

Commutativity effect takes place when an order in which actions $a$ and $b$ are performed does not make an impact upon the result of their application (for example, when $a$ and $b$ are program statements $x := y + x$, $z := y + 1$.

The frame $\mathcal{F}_{\text{comm}}$ is both ordered and injective.
Programs with reversible instructions

A basic instruction $a$ is called reversible on a frame $\mathcal{F} = \langle S, s_0, R \rangle$ if for every data state $s$ there exists a finite sequence of instructions $h_s = b_1, b_2, \ldots, b_n$ such that $R^*(h_s a, s) = s$. 

Consider now the complexity of the equivalence checking problem for programs with reversible instructions.
A basic instructions $a$ is called reversible on a frame $\mathcal{F} = \langle S, s_0, R \rangle$ if for every data state $s$ there exists a finite sequence of instructions $h_s = b_1, b_2, \ldots, b_n$ such that $R^*(h_s a, s) = s$.

Consider now the complexity of the equivalence checking problem for programs with reversible instructions.
Programs with reversible instructions

«Short» loops

Instructions:  
- $a$: $x = x + 1 \mod y$
- $b$: \{ $x = 0$; $y = y + 1$ \}

$F_1 = \langle S_1, s_0^1, R_1 \rangle$, where

- $S_1 = \mathbb{N} \times \mathbb{N}$,
- $s_0^1 = \langle 0, 0 \rangle$,
- $R_1(a, \langle i, j \rangle) = \langle i, j + 1 \mod i \rangle$ and $R_1(b, \langle i, j \rangle) = \langle i + 1, 0 \rangle$. 

Theorem 6. The non-equivalence checking problem for the frame $F_1$ is NP-complete.
Programs with reversible instructions

«Short» loops

Instructions: $a$: $x = x + 1 \mod y$
               $b$: $\{x = 0; \ y = y + 1\}$

$\mathcal{F}_1 = \langle S_1, s_0^1, R_1 \rangle$, where

- $S_1 = \mathbb{N} \times \mathbb{N}$,
- $s_0^1 = \langle 0, 0 \rangle$,
- $R_1(a, \langle i, j \rangle) = \langle i, j + 1 \mod i \rangle$ and $R_1(b, \langle i, j \rangle) = \langle i + 1, 0 \rangle$.

**Theorem 6.** The non-equivalence checking problem for the frame $\mathcal{F}_1$ is NP-complete.
Programs with reversible instructions

Free group of rank 1

Instructions: \(a\): \(x = x + 1\),
\(b\): \(x = x - 1\).

\(\mathcal{F}_2 = \langle S_2, s_0^2, R_2 \rangle\), where

- \(S_2 = \mathbb{Z}\),
- \(s_0^2 = 0\),
- \(R_2(a, i) = i + 1\) and \(R_2(b, i) = i - 1\).
Programs with reversible instructions

Free group of rank 1

Instructions: $a$: $x = x + 1$,  
$b$: $x = x - 1$.

$\mathcal{F}_2 = \langle S_2, s_0^2, R_2 \rangle$, where

- $S_2 = \mathbb{Z}$,
- $s_0^2 = 0$,
- $R_2(a, i) = i + 1$ and $R_2(b, i) = i - 1$.

**Theorem 7.** The equivalence checking problem for the frame $\mathcal{F}_2$ is PSPACE-complete.
Free group of rank 2

Instructions: \( a, \ a^{-1}, \ b, \ b^{-1} \)

\( \mathcal{F}_3 = \langle S_3, s_0^3, R_3 \rangle \), where

- \( S_3 \) — free group generated by the elements \( a \) and \( b \),
- \( s_0^3 \) — the neutral element of the group,
- \( R_3(x, s) = x \circ s \).
Programs with reversible instructions

Free group of rank 2

Instructions: $a, a^{-1}, b, b^{-1}$

$F_3 = \langle S_3, s_0^3, R_3 \rangle$, where

- $S_3$ — free group generated by the elements $a$ and $b$,
- $s_0^3$ — the neutral element of the group,
- $R_3(x, s) = x \circ s$.

**Theorem 8.** The equivalence checking problem for the frame $F_3$ is EXPTIME-complete.
Programs with reversible instructions

Free Abelian group of rank 2

Instructions: $a$: $x = x + 1$, $a^{-1}$: $x = x - 1$, $b$: $y = y + 1$, $b^{-1}$: $y = y - 1$

$\mathcal{F}_4 = \langle S_4, s_0^4, R_4 \rangle$, where

- $S_4 = \mathbb{Z} \times \mathbb{Z}$,
- $s_0^4 = \langle 0, 0 \rangle$,
- $R_4(a, \langle i, j \rangle) = \langle i + 1, j \rangle$ and $R_2(b, \langle i, j \rangle) = \langle i, j + 1 \rangle$. 

Theorem 9. [Gurevich, Letichevsky] The equivalence checking problem for the frame $\mathcal{F}_4$ is undecidable.
Programs with reversible instructions

Free Abelian group of rank 2

Instructions: $a : \ x=x+1, \quad a^{-1} : \ x=x-1,$
$b : \ y=y+1, \quad b^{-1} : \ y=y-1$

$F_4 = \langle S_4, s_0^4, R_4 \rangle$, where

- $S_4 = \mathbb{Z} \times \mathbb{Z}$,
- $s_0^4 = \langle 0, 0 \rangle$,
- $R_4(a, \langle i, j \rangle) = \langle i + 1, j \rangle$ and $R_2(b, \langle i, j \rangle) = \langle i, j + 1 \rangle$.

**Theorem 9.** [Gurevich, Letichevsky] The equivalence checking problem for the frame $F_4$ is undecidable.
A basic statement \( a \) is called a reset instructions on a frame \( \mathcal{F} = \langle S, s_0, R \rangle \) if \( R(a, s_1) = R(a, s_2) \) holds for every pair of states \( s_1, s_2 \).
A basic statement $a$ is called a reset instructions on a frame $\mathcal{F} = \langle S, s_0, R \rangle$ if $R(a, s_1) = R(a, s_2)$ holds for every pair of states $s_1, s_2$.

It is clear that if $a$ is a reset instruction such that $R(a, s) = s_0$ then every instruction is reversible on the frame $\mathcal{F}$. 

Programs with reset instructions
A basic statement \( a \) is called a reset instruction on a frame \( \mathcal{F} = \langle S, s_0, R \rangle \) if \( R(a, s_1) = R(a, s_2) \) holds for every pair of states \( s_1, s_2 \).

It is clear that if \( a \) is a reset instruction such that \( R(a, s) = s_0 \) then every instruction is reversible on the frame \( \mathcal{F} \).

Now we turn to the equivalence problem for the frames with rest instructions.
Programs with reset instructions

Free semigroups with right zeros

Instructions:  
\[ a : \ x = x + 1 \]
\[ b : \ x = 0 \]

\[ \mathcal{F}_5 = \langle S_5, s_0^5, R_5 \rangle, \text{ where} \]
\[ S_5 = \mathbb{N}, \]
\[ s_0^5 = 0, \]
\[ R_5(a, i) = i + 1 \quad \text{and} \quad R_5(b, i) = 0. \]
Programs with reset instructions

Free semigroups with right zeros

Instructions:  
\[ a : \ x = x + 1 \]
\[ b : \ x = 0 \]

\( \mathcal{F}_5 = \langle S_5, s_0^5, R_5 \rangle \), where

- \( S_5 = \mathbb{N} \),
- \( s_0^5 = 0 \),
- \( R_5(a, i) = i + 1 \) and \( R_5(b, i) = 0 \).

**Теорема 10.** The equivalence checking problem for the frame \( \mathcal{F}_5 \) is PSPACE-complete.
Programs with reset instructions

Free commutative semigroups with right zeros

Instructions: \( a_1 : \ x=x+1, \ a_2 : \ y=y+1, \)
\[ b : \ \{x=0; \ y=0\} \]
\( F_6 = \langle S_6, s^6_0, R_6 \rangle, \) where

- \( S_6 = \mathbb{N} \times \mathbb{N} \),
- \( s^6_0 = \langle 0, 0 \rangle \),
- \( R_6(a_1, \langle i, j \rangle) = \langle i + 1, j \rangle, \ R_6(a_2, \langle i, j \rangle) = \langle i, j + 1 \rangle, \)
\( R_6(b, \langle i, j \rangle) = \langle 0, 0 \rangle. \)
Programs with reset instructions

Free commutative semigroups with right zeros

Instructions: \( a_1 : x = x + 1, \ a_2 : y = y + 1, \)
\( b : \{x=0; \ y=0\} \)
\( \mathcal{F}_6 = \langle S_6, s_0^6, R_6 \rangle, \) where

- \( S_6 = \mathbb{N} \times \mathbb{N}, \)
- \( s_0^6 = \langle 0, 0 \rangle, \)
- \( R_6(a_1, \langle i, j \rangle) = \langle i + 1, j \rangle, \ R_6(a_2, \langle i, j \rangle) = \langle i, j + 1 \rangle, \)
- \( R_6(b, \langle i, j \rangle) = \langle 0, 0 \rangle. \)

**Theorem 11.** The equivalence checking problem for the frame \( \mathcal{F}_6 \) is PSPACE-complete.
Programs with reset instructions

A product of free semigroups with right zeros

Instructions: $a_1 : x=x+1$, $a_2 : y=y+1$,  
$b_1 : x=0$, $b_2 : y=0$,

$\mathcal{F}_7 = \langle S_7, s_0^7, R_7 \rangle$, where

- $S_7 = \mathbb{N} \times \mathbb{N}$,
- $s_0^7 = \langle 0, 0 \rangle$,  
- $R_7(a_1, \langle i, j \rangle) = \langle i + 1, j \rangle$, $R_7(a_2, \langle i, j \rangle) = \langle i, j + 1 \rangle$,  
  $R_7(b_1, \langle i, j \rangle) = \langle 0, j \rangle$, $R_7(b_2, \langle i, j \rangle) = \langle i, 0 \rangle$.  

Theorem 12. [Petrosyan, Godlevsky]
The equivalence checking problem for the frame $\mathcal{F}_7$ is undecidable.
Programs with reset instructions

A product of free semigroups with right zeros

Instructions:  $a_1 : x=x+1$,  $a_2 : y=y+1$,

$\begin{align*}
b_1 : x=0, & \quad b_2 : y=0,
\end{align*}$

$F_7 = \langle S_7, s^7_0, R_7 \rangle$, where

$\begin{align*}
\text{➢ } S_7 &= \mathbb{N} \times \mathbb{N}, \\
\text{➢ } s^7_0 &= \langle 0, 0 \rangle, \\
\text{➢ } R_7(a_1, \langle i, j \rangle) &= \langle i + 1, j \rangle, \quad R_7(a_2, \langle i, j \rangle) = \langle i, j + 1 \rangle, \\
R_7(b_1, \langle i, j \rangle) &= \langle 0, j \rangle, \quad R_7(b_2, \langle i, j \rangle) = \langle i, 0 \rangle.
\end{align*}$

**Theorem 12.** [Petrosyan, Godlevsky] The equivalence checking problem for the frame $F_7$ is undecidable.
Open problems

1. Suppose that $Time(f(n))$ is a complexity class such that $NLOG \subseteq Time(f(n))$. Is there any frame $\mathcal{F}$ such that the (non)-equivalence problem for $\mathcal{F}$ is $Time(f(n))$-complete?
Open problems

1. Suppose that $Time(f(n))$ is a complexity class such that $NLOG \subseteq Time(f(n))$. Is there any frame $\mathcal{F}$ such that the (non)-equivalence problem for $\mathcal{F}$ is $Time(f(n))$-complete?

2. What are the necessary conditions for deciding the equivalence checking problem for $\mathcal{F}$ in polynomial time?

3. What are characteristic algebraic properties of semigroups $\mathcal{F}$ that are specified by finite-state 2-DMs?
Thank you for your patience